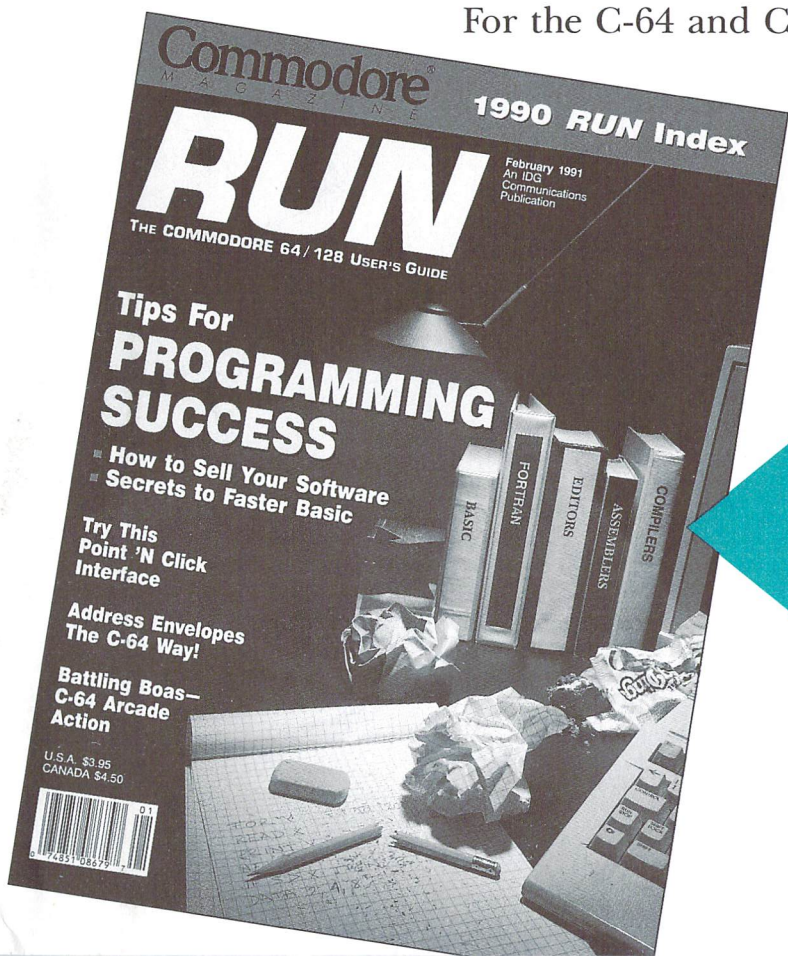


January/February 1991 Edition

RE₌₌RUN

RUN Programs on Disk

For the C-64 and C-128



*Plus:
Extra
Bonus
Programs!*

Introduction

January/February '91 ReRUN

WELCOME TO THE JANUARY/FEBRUARY 1991 EDITION of ReRUN. As many of you may realize, *RUN* will be published on a bi-monthly schedule beginning with the January/February issue, which means we will publish six issues in 1991. This change reflects our need to adjust to the reduction in size of a once robust 64/128 market.

Subscribers to ReRUN will be unaffected by the change in *RUN*'s frequency, since ReRUN will continue to be produced throughout the year on its normal bi-monthly schedule. I believe that it's important for all of our ReRUN customers to know that ReRUN will contain more bonus programs to compensate for the reduction in the number of programs published in *RUN*. By increasing the number of bonus programs in each edition of ReRUN, we have the ability to deliver the healthy mix of applications, utilities and games that you've come to expect from us.

We begin this time with Buttons, a truly useful C-64 utility from the January/February issue of *RUN*. Buttons gives your computer the look and feel of a point-and-click user interface. Using a Commodore 1351 mouse plugged into joystick port #1, you can move a pointer around the screen. As the pointer passes over buttons placed about the screen, various messages appear. The program is useful as a routine for programmers of all skill levels as a framework for their own mouse-driven operating systems for the 64.

Have you ever tried to print the return and delivery addresses on an envelope using your computer's printer? Chances are good that you probably got one good envelope printed after ruining about four or five. Well, Envelope Addresser to the rescue! Also hailing from the January/February issue of *RUN*, Envelope Addresser gives your printer the ability to print both "to" and "from" addresses on business envelopes without a hitch. Give it a try, as I'm sure that it'll convince you that it's probably the easiest way yet to create envelopes with your printer.

When you're ready for a little rest and relaxation, boot up Battling Boas, the next program on the disk. This 100% machine language action/arcade game pits you against either an opponent or the com-

puter for fast-paced strategy action. The object is to either box your opponent in, or force them to collide with the trail left behind each player. It's a challenge at any of its multitude of levels of difficulty.

Bonus programs abound in this edition, and the first is Questionnaire 64/128, written by long-time *RUN* contributor, Dr. Hugh McMenamin. The good doctor came up with the idea of a questionnaire generator and its accompanying Analyzer program for diagnosing the answers to the questions. Whether used for fun or serious work, Questionnaire provides a good method of collecting and analyzing data for everything from planning menus at social functions to organizing after-school activities.

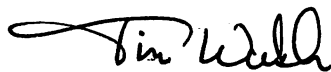
Bring home the flavor of ancient times with Vanquish!, a two-player C-64 checkers-like game. All that you need to play is two joysticks and the desire to outwit your opponent.

Next on the list is SAR (Some Assembly Required) 128, a commercial-quality machine language assembler for the C-128 written by *RUN*'s long-time contributor, Jim Borden. Designed for anyone with a hankering for machine language programming in mind, SAR 128 also includes Jim's AFCO utility, previously published in *RUN*. If you've never tried your hand at machine language programming, give SAR 128 a try and you're certain to find it a comprehensive way to flex programming muscles.

80-Column 64 creates just that—an 80-column mode for the C-64 on the 64's 40-column screen. It's the next best thing to owning a C-128 with its built-in 80-Column mode. 80-Column 64 is followed by Random Sound 128, designed for the "real" 80-Column mode on the C-128. You use it to rapidly create a variety of sounds and sound effects using all three voices of the C-128 SID chip.

A Re*RUN* disk just wouldn't be complete without a Tony Brantner game, and this disk is no exception. You'll find that we've included Tony's Laser Math to serve as the final program on this disk. With its three levels and nine speed settings, everyone from first graders to college students can hone their math skills with Laser Math.

Well, that concludes this edition. Stay tuned, and I'll be back in two months with another exciting edition of Re*RUN*.



Technical Manager
RUN Magazine

Directory

PAGE	DOCUMENTATION	DISK FILENAME	FILE TYPE
		*MENU 128 _____	BASIC
		MENU 64 _____	BASIC
1	BUTTONS	BUTTONS DEMO _____	BASIC
		BUTTONS.O _____	ML
		SAMPLE.O _____	ML
		MOUSE.POINTER _____	ML
		MAKE POINTER _____	BASIC
		MAKE SAMPLE _____	BASIC
		MAKE BUTTONS _____	BASIC
5	ENVELOPE ADDRESSER	ENVELOPE ADDR _____	BASIC
6	BATTLING BOAS	BOOT BOAS _____	BASIC
		BOAS _____	ML
		MAKE BOAS _____	BASIC
8	\ QUESTIONNAIRE	QUESTIONNAIRE _____	BASIC
		ANALYZER _____	BASIC
11	\ VANQUISH!	VANQUISH! _____	BASIC
12	\ * SAR 128	SAR 128 _____	BASIC
		%SAR5.42 DOLO _____	ML
		%SAR5.42 DOOO _____	ML
		%AFCO 5.42 _____	ML
27	\ 80 COLUMN 64	80 COLUMNS _____	BASIC
		DEMO1-80 _____	BASIC
		DEMO2-80 _____	BASIC
29	\ * RANDOM SOUNDS 128	RANDOM SOUNDS128 _____	BASIC
31	\ LASER MATH	LASER MATH _____	BASIC

* — C-128 mode only

— Requires GEOS

\ — Bonus program

Before you run a program, carefully read the documentation that pertains to it.

How To Load

LOADING FROM MENU

To get started, C-64 users should type LOAD "MENU 64",8 and press the return key. When you get the Ready prompt, the menu is loaded and you should type RUN to see a list of the programs on your disk. C-128 users need only press the shift and run-stop keys. When all the programs are displayed on the screen, you can run the one you select by pressing a single key.

LOADING FROM KEYBOARD

If you do not wish to use the menu program, follow these instructions.

C-64: To load a C-64 program written in Basic, type: LOAD "DISK FILENAME",8 and then press the return key. The drive will whirl while the screen prints LOADING and then READY, with a flashing cursor beneath. Type RUN and press the return key. The program will then start running. To load a C-64 program written in machine language (ML), type: LOAD "DISK FILENAME",8,1

C-128: All C-64 programs can be run on the C-128 as long as your computer is in C-64 mode. All C-128 programs are clearly labeled on the directory page. Your C-128 *must* be in C-128 mode to run these programs. To load a C-128 mode program, press the F2 key, type the disk filename and then press the return key. When the program has loaded, type RUN.

MAKING COPIES OF ReRUN FILES

Many programs on your ReRUN disk have routines that require a separate disk onto which the program writes or saves subfiles. To use these programs, you must first make a copy of the original program onto another disk that has enough free space on it to hold these newly written subfiles.

It's simple to make a copy of a Basic program. Just load it into your computer as outlined above, and then save the program back onto a separate disk that has plenty of free space for extra files.

Copying an ML program is not so simple. You cannot simply load and save an ML program; you'll need to use a disk-backup utility program, such as the one on your Commodore Test Demo disk.

Button Up Your Programs

By Kevin Smotherman

DTLF (DOES THIS LOOK FAMILIAR)? HAT (How About This)?
1 = Or 2 = Maybe 3 = You 4 = Do 5 = This?

I've used all these methods to prompt for user input in my programs, but they're so cumbersome and cryptic. I finally got fed up with them and invested a few Saturday afternoons in creating a better way. Buttons—a flexible user interface system that is friendly enough for even a novice computerist—is the result.

Buttons has made my programs more professional looking, easier to use, and typically reduces the size of my Basic programs by 50 percent or more. And since it's written completely in assembly language, it executes with blinding speed.

If you don't want to convert to a new user-input routine, you don't have to. Buttons is a point-and-click user interface, using your proportional mouse in a dialog-box fashion. Integrate the two, using Buttons for menu selections and your own input routine for data that requires typing.

WHAT'S A BUTTON?

A "button" is a defined object that's used to get selections from the user. The appearance and function of each button on the "button screen" are defined by table structures. A button consists of three parts, each of which may or may not be switched on for any particular button. The first part (which must be present) is the "button text," a group of characters (including graphics characters) that's displayed on the "face" of the button. Think of the button as a key on your Commodore keyboard; the button text would be the character(s) on the top of the key.

The second part of a button is the border, a rectangle that surrounds the text. You can determine whether or not it will be present.

The last part of a button is the shadow, which appears as reverse video spaces to the left of and below the button. It gives a button a

three-dimensional effect, similar to the shadow you see on GEOS dialog-box buttons.

BUTTON TABLE

Each part of a button may be a different color, as defined in the button table, which is a list of button definitions, preceded by a count of the buttons in that table. When Buttons displays a button screen, it doesn't erase any part of the screen; it just puts the buttons on top of whatever is already there.

Each button is described by a string that may be up to 40 characters long. Whenever the mouse sprite pointer is positioned over a button, this description string is displayed in a window. The location and color of this single-line window is user definable.

Each button also has a "flash control," which comes into effect when you put the mouse pointer over a button and press the mouse button. The button may momentarily change colors and then change

Table 1. Button table format.

Byte	Bits	Function
1 0-7	0-255	Number of buttons in this table
2 0-4	0-31	Button Y screen coordinate (0 to 23)
6	64	Shadow display; on = suppress shadow
7	128	Border display; on = suppress border
3 0-5	0-63	Button X screen coordinate (0 to 39)
7	128	Flash control: on = change, off = flash
4 0-5	0-63	Button width
5 0-3	0-15	Border color
4-7	0-15	Button text color (0 to 15 × 16)
6 0-3	0-15	Shadow color
4-7	0-15	Flash color (0 to 15 × 16)
7 0-7	0-255	Button text pointer, low byte
8 0-7	0-255	Button text pointer, high byte
9 0-7	0-255	Button description pointer, low byte
10 0-7	0-255	Button description pointer, high byte
...	...	More buttons
...	...	More tables

back to its original color, or it may remain the new color until selected again, or until Buttons redraws that button table.

Sound complicated? It's not. Once you've tried Buttons, you'll never go back to another system!

The format of a button table is shown in Table 1. Note that any number of button tables may be pushed together back to back, and you can tell Buttons which to activate. You may define button tables anywhere in memory as long as everything is consecutive. The easiest way to define button tables is with an assembler, but you can also poke them into memory or build them as data files and load them where you want.

The "button width" includes the border (two characters) and the shadow (one character). When defining the button width, allow for the border/shadow only if the button is being defined with the border/shadow option.

The "button text pointer" is a two-byte low/high pointer to a text string of length *exactly* equal to the button width minus 2 (if a border is used) and/or minus 1 (if a shadow is used).

The "button description pointer" is a two-byte low/high pointer

Table 2. Memory locations to display buttons.

Location	Function
679	Button selected (1 to number of buttons)
680	Button table number to display/activate (1 to number of tables)
681,682	Vector to routine that displays a button table
683,684	Vector to routine that activates a button table
690	Description window Y screen coordinate (0 to 23)
691	Description window X screen coordinate (0 to 39)
692	Description window width (0 to 40; 0 = no window)
693	Description window color (0 to 15; add 128 to use reverse video)
828,829	Pointer to start of last displayed button table
833,834	Pointer to start of first button table
49152	Address of routine that initializes the Buttons environment

to a text string that is zero-delimited (a zero byte ends the string). If this text string is wider than the window for button descriptions, it will be truncated at the window length.

So, now you know how to define buttons, how to group them in a button table and how to group tables consecutively for Buttons to display. To display these buttons, refer to the memory map in Table 2 that outlines important memory locations to use.

USING BUTTONS

To use Buttons, start by loading and running Menu 64, then select **BUTTONS DEMO**.

To copy the program files **BUTTONS.O**, **MOUSE.POINTER** and **SAMPLE.O** to a work disk, simply load and run **MAKE SAMPLE**, **MAKE POINTER** and **MAKE BUTTONS** with a work disk in the drive. Then copy **BUTTONS DEMO** to the same work disk.

MOUSE.POINTER is simply a sprite definition program. You can define any sprite you want—hi-res or multi-color—provided it is sprite 0, and you should set address 2040 to indicate what 64-byte group you want to use to store the sprite. I recommend block 11 (starts at address 704). A complete discussion of sprites is beyond the scope of this article, but the sample program contains an example of how to set up a sprite for Buttons to use.

Next poke or load in your button table definitions. After this, poke in the values to position the description window. **SYS 49152** will now set up the Buttons program to use these parameters and will initialize memory locations 833,834 to point to the first free byte past the actual Buttons driver code. You may change it to point wherever you want, though.

To display a button table, type in **POKE 680** with the button table number to display, then type **SYS PEEK(681) + PEEK(682)*256** to display the button screen. To activate the buttons and allow the user to select one, type **POKE 680** with the button table number to activate (if you need to), and **SYS PEEK(683) + PEEK(684)*256** to activate it.

After the user selects a button, the **SYS** call will return and memory address 679 will reflect which button was selected (1 to number of buttons). To reactivate the same table, just repeat the **SYS PEEK(683) + PEEK(684)*256**.

If you have a button defined with the Shadow option turned on, the shadow is what will be flashed by the Buttons driver. If the Border option is on and the Shadow option is off, then the border

will be flashed. If both shadow and border are off, then the button text itself is flashed.

With these simple building blocks, you'd be surprised at the complexity of menu-type structures you can create. And, best of all, each one is just a point and click away!

RUN it right: **C-64; printer**

Envelope Addresser

By Kevin McDonald

ENVELOPE ADDRESSER IS JUST the program you need when it's time to get your mail out. This nifty little program lets you print addresses—both forwarded (To) and return (From) addresses—on standard letter and legal envelopes just by using the C-64's function keys. Your return address is saved in a sequential file on disk, then, each time Envelope Addresser runs, it's loaded automatically.

Load and run ENVELOPE ADDR from Menu 64. When you run the program for the first time, the message "Loading From Address" flashes across the screen. Then, because there is no From address yet, the "Entering a New From Address" screen follows.

Place a formatted work disk in the drive for saving your "To" and "From" addresses.

Enter your From address one line at a time, pressing the return key at the end of each line. The first character on each line is a set of quotation marks provided by the program; it's under the cursor when the line first appears. Press the cursor-right key once before typing any line that contains a comma or other punctuation mark. If you don't, you may get an Extra Ignored error message and lose everything typed in the line after the punctuation mark. Don't worry if you forget, though—the line can be corrected after you finish entering the rest of the address. You can enter a blank line by tapping the return key when the line first appears. To end the

address, type £ as the first character of the line following the address and then press the return key.

THE MENUS

There are three Envelope Addresser menus. All menu items are selected by pressing the appropriate function key.

The From Address menu: F1 lets you quickly change the first line of the From address—useful when several people have the same address. F2 inserts a new line into the address; F3 enters a completely new address; F4 deletes a line from the address; and F5 makes corrections to any line in the address. F6 is for saving the From address, as it appears on the screen, to disk. Save the From address the first time you enter it and any time you make permanent changes to it. F7 lets you access the To Address menu.

The To Address menu: The F2, F3, F4 and F5 keys are the same as in the From Address menu, while F1 and F6 are no longer available. F7 advances to envelope printing.

To print, put an envelope in the printer and press any key. When printing is done, the Print Options menu appears.

The Print Options menu: F1 prints another envelope; F3 lets you make corrections in both addresses; F5 enters a new To address; and F7 exits to Basic.

Now, with this program—and a little help from your friendly postman—your mail is sure to be delivered correctly!

RUN it right: **C-64; one or two joysticks**

Battling Boas

By Steve Harter

THE GOAL OF BATTLING BOAS is to add as many segments as possible to your snaky line of blocks curling around the screen. When you hit an existing block, either yours or an opponent's, you're out for the round, and, if there are only two players (including the computer), the round is over. While avoiding collisions yourself, of course, you should try to force your opponent(s) into collisions.

The game is designed for either two people, one person and the computer or two people and the computer. When only one person is playing, the joystick should be plugged into port 2. The program is written completely in machine language.

Activate the program by selecting **BOOT BOAS** from Menu 64. When the menu screen appears, you must choose the configuration of players and the other game options (described below) that you want. Simply move among the menu items by pressing the joystick forward or backward, and, if necessary, change your choices by pressing it left or right.

When the game configuration is all set, press the firebutton to start play. (It won't start if you're playing alone against the computer with your joystick plugged into port 1.)

Soon the game screen will appear, and then a block for each player at a random location within it. The red block belongs to player number 1, the green to player number 2 and the blue to the computer.

Move your block by pressing your joystick in the corresponding direction. As the block moves, another block will be left in its place, and then in each location it passes through. Thus, the "tail" of your "snake" will keep growing, until it reaches the length chosen at the menu screen. The snake cannot move backward.

The last player to avoid a collision is the winner of the round. The game continues for the number of rounds chosen at the menu, and the game winner is the player with the highest score at that point. When the game is over, press the firebutton to return to the menu screen.

You can pause the action at any time by pressing the run-stop key; then press it again to continue. During a pause, you can quit the current game and return to the menu screen by pressing the Q key.

THE OPTIONS

Border: If set to "on," the border of the game screen is gray and impenetrable; if set to "off," the border is brown, signifying that the snakes can wrap around from one side of the screen to the other.

Speed Up: If set to "on," the snakes gradually move faster; if set to "off," their speed remains the same.

Random Blocks: If set to "on," extraneous blocks appear on the screen, obstructing your way; if set to "off," no such blocks appear.

Tail Length: Specifies to what length, in blocks, the snakes' tails will grow.

Rounds: Specifies the number of rounds in a game.

Speed: Specifies the snakes' starting speed. If you don't opt for speeding up, this will be their speed for the entire game.

Fire: These options specify what happens when you press the firebutton. "Off" results in no reaction; "inc speed" doubles your snake's speed; "hyperspace" makes your snake disappear, then reappear elsewhere, still moving in the same direction; "leave spaces" makes blank spaces appear in your snake's tail, rather than blocks; "skip" lets your snake "pass under" a single block, then "come up" on the other side. Only one of these options is in effect at a time.

SCORING

Each block you add to your snake is worth one point, and if you win a round you receive a bonus of 200 points. In addition, the numbers 20, 40 and 80 will appear randomly on the screen, and if you can "collect" one, you'll get 20, 40 or 80 points.

There are also little diamond-shaped characters that appear randomly on the screen. If you collect one of them, it, in turn, makes two other characters appear—either two more diamonds or two 20s, 40s or 80s. The diamond is not worth any points in itself.

Now, get busy and prove just how clever you are!

RUN it right: C-64 or C-128 (in 40- or 80-Column mode)

Twenty-five Questionnaire

By Hugh McMenamin

CONDUCTING A GROUP SURVEY is often time-consuming and laborious, but now you can generate the forms and tabulate the results quickly and easily with my Questionnaire and Analyzer programs. Questionnaire prepares a form containing blanks for identification data and up to 25 questions to be answered yes or no with a check mark. When you enter the information from the question-

naires into the Analyzer database, it prints out data on individuals who checked the yes box for any chosen question.

Be certain that you have a formatted work disk handy. To use either Questionnaire or Analyzer, just load and run either Menu 64 or Menu 128 and select the one you want. Once either program is activated be sure to insert the work disk in the drive, since both Questionnaire and Analyzer read and write files to the disk. Naturally, the work disk is needed for subsequent sessions with the program.

QUESTIONNAIRE

If you don't begin the session by loading a previously created file, the questionnaire form prompts you to enter your title, followed by name, address, city, state, ZIP and an optional special category that you can define, such as class, parish or age. The actual questions, which you also type in, must be less than 60 characters long. Provision has been made for correcting errors in the questions and the special category name after entry. If there are no corrections, the questions are stored on the work disk for later recall and use by Analyzer. The name of the file they're stored in gets a q. prefix.

You can also revise a form after recalling it from disk. If the revision includes a different number of questions, you'll have to save it in a new file, or all the data previously entered in Analyzer will be rendered useless.

Your questionnaire can be printed out on any printer. If it's a daisywheel printer, the variable BX\$ in the program should be changed to brackets, [], or parentheses, (). After printing, you can make as many copies of the questionnaire as you like on a photocopier.

ANALYZER

Analyzer is a database program specifically designed for use with Questionnaire. When it's run, a menu with eight options appears. First you must choose option 1, to load a question file from the work disk used with Questionnaire. If there's no previous index file for those questions on the disk, Analyzer creates a sequential file with an n. prefix to hold the number of records stored and a relative file with a d. prefix for data in the records.

After loading the question file from the work disk you created, select menu option 2 to enter data. You'll be asked for each respondent's last name, first name, address, city, state (two-letter abbreviation only), ZIP code and phone number. If you included a

special category in the questionnaire, you'll be asked for the response to that as well. If not, press return to continue. When you're done entering the identification data, you'll have an opportunity to select and correct entries. Then you'll be asked for the yes/no answers to the questions in the question file. Again, you can select and change any incorrect entries. When you indicate the information is correct, it's saved to the relative file on the disk.

Menu option 3 lets you review any file by number. Option 4 lists the last name of each respondent in the file chosen from option 3. Both these options are for information only.

Option 5, the most important, prints a list of yes responders to a question either on the screen or on both screen and paper. First, the program displays all the questions in your file, each identified by a reversed letter. You select the question you want by entering its letter. Then you choose any identification data—such as address or phone number—you want printed with each name; it will appear after the name in the order of selection.

Next, the program will ask if you want a hardcopy. If yes, then press y (lowercase) to send the output to both screen and printer; otherwise it will be sent to the screen only.

Option 6 is the master correction section of Analyzer. It displays the entire list of names in a file on the screen, along with record numbers. To change a record, enter its number at the prompt, then, when the record appears, enter the number of the line you want to change and press return. When the line appears on the screen, the cursor will be poised over the first character. Type your changes on the line and press return. Finally, the program will display the file as corrected and ask if it's okay. If so, answer y and the file will be saved. Press n (or any other key) to reject the changes.

Option 7 displays the number of files you've entered. If you decrease this number, the data in the higher-numbered files will be overwritten when new data is entered.

Option 8 exits Analyzer.

These programs provide an easy way to survey opinion or solicit help. In either case, if you don't get enough yeses, just rephrase the questions!

RUN it right: **C-64; two joysticks**

Vanquish!

By Kurt Ehland

DURING THE TIME OF MIDDLE EARTH, the evil sorcerer Zepher built a magical arena where armies of monsters battled for survival. Eventually, only two armies—the red dragons and the hippogriffs—were left. *Vanquish!* replays the battle between those armies—the last battle in the arena.

Vanquish! is a two-person game played with joysticks. The joystick in port 1 controls the red dragons, and the joystick in port 2 controls the hippogriffs. Load and run Menu 64, then select **VANQUISH!** to begin.

After the title screen, you'll be asked if you want obstacles on the gameboard. They make play more challenging, but I don't recommend them for beginners. Then the arena, an 8 by 8 grid, will be drawn, with the armies arrayed at either end. Each army consists of eight monsters.

Moves are controlled by dice, which are automatically rolled by the computer. To "roll," you actually press the joystick firebutton to stop the die from rolling.

You and your opponent must start by rolling to see who moves first; the higher roll wins. When it's your turn to move, roll a number of spaces, then choose the monster in your army you want to activate. Here's how to choose: Notice the four dark squares in the corners around the first-row, first-column space in the grid. Using your joystick, slide this arrangement of squares onto the space that holds the monster you want, then press the firebutton.

You can move the monster either forward or sideways the number of spaces you rolled. A buzzer will sound if it moves the wrong number of spaces, tries to cross its own path, runs off the grid, or runs into an obstacle or other monster. When the buzzer sounds five times, your turn ends.

To banish an enemy monster from the arena, your monster must land in its space by moving the exact number of spaces you've rolled. For example, if you've rolled five, you must move four spaces, then

land on the enemy. The winner is the first player to Vanquish! the entire opposing army.

RUN it right: C-128 (in 40- or 80-Column mode)

The SAR 128 Assembler

By Jim Borden

SAR STANDS FOR some assembly required, and this easy-to-use 128 Assembler is useful not only to the beginner but to the intermediate and advanced machine language programmer. SAR's source code is entered as a Basic program, so you have all the entry and edit features of Basic 7.0 and a familiar environment. In fact, you can enter SAR source code without SAR in memory. The only restriction while entering source code lines is that "fields" (opcodes, addresses, labels, remarks) must be separated by at least one space.

SAR assembles directly to memory while the source is in memory. This provides interactive editing much like that of Basic. Assembly is usually sent to RAM bank 0, but can also go to RAM bank 1 or a disk file. The source code can be up to about 45K long, which will assemble to roughly 6K of machine language object code (depending on the size and number of comments used). Source files can also be linked for very large programs.

GETTING STARTED

To execute SAR128, just select SAR 128 from Menu 128. Both SAR 128 and AFCO are then activated.

Below is a short program that prints a message to the screen. Turn on your C-128 and run the SAR Boot program. It's always a good idea to type NEW before starting to enter your source code. Now enter AUTO 10 and press return. Enter the program exactly as shown, including the deliberate errors. They'll let me describe how to correct errors.

```
10 ;PRINT YOUR MESSAGE
20 *=0B00 ;ASSIGN PC
30 CHROUT=$FFD2
```

```

40 ZSTART LDX #0
50 LOOP1 LDA TEXT,X
60 BEQ EXIT ;ZERO BYTE=>END
70 JSR CHROUT
80 INX GET NEXT BYTE
90 BNE LOOP1 ;FORCE
100 EXIT RTS ;DONE→TO BASIC
110 TEXT .BYTE $93,"MY NAME IS SAR",0
120 ZZEND! ;SHOWS LAST BYTE ADR+1

```

When you've entered all the lines, press return, type AUTO and press return again. This will cancel the Auto line-number function. Type L and press return, and you should see the program listed with the fields in neat columns.

To assemble with SAR, just type A and press return. (All SAR commands are one character long.) Remember, there are several intentional errors in this program.

You should now see:

```

INVALID ADDRESS IN LINE 00020.
NO ORIGIN IN LINE 00040.

```

Since the No Origin error is "fatal," assembly stops at this point. Nothing has been assembled yet, since no location was given as a starting point. Line 20 does give a starting point, but the address is 0B00, which can't be a decimal address (no prefix means decimal in SAR). Line 30 just sets the value of CHROUT to \$FFD2, so it's OK, but line 40 tries to assign the label ZSTART to the current PC location. At that point SAR stops assembly. To correct both problems, just list line 20, change the first 0 to a dollar sign (\$B00) and press return. Then type A and return to try the assembly again.

More errors! This time you should see:

```

INVALID ADDRESS MODE IN LINE 00080.
INVALID LABEL IN LINE 00120.

```

```

LABEL NOT FOUND IN LINE 00080.
INVALID LABEL IN LINE 00120.

```

Most errors will be reported on both passes, as you can see here. Line 80 reports a different error on each pass, but the cause is the same. All that's required to fix that line is inserting a semicolon (;), to begin a remark, in front of the comment GET NEXT BYTE. List line 80 and correct it now.

List line 120 and delete the exclamation point from the label, since only letters, numbers and the apostrophe (') are valid label characters. Then assemble the source again. If you haven't made any unintentional errors, the code will assemble and you'll see the Assembled Okay message on the screen.

The code is now ready to use. It's a good idea to save your source before you assemble it, although we didn't do it here because of the intentional errors. Save it now, in case there's an error in the machine language program. Such an error may result in a lock-up and loss of the program!

To test the machine language code, type BANK15:SYS2816 and press return. You should see the message in line 110 printed at the top of your screen. You can change the message if you like. Just be sure to put it in quotes and include a comma and zero after the text. The zero is used by the program to detect the end of the message. Assemble the modified program and then use SYS 2816 to see the results.

To see the labels (or symbols, as they're often called) used by the program, type T (for table) and press return. This will print the label table to the screen. If you're using a 40-column monitor, the labels will be printed two per line. If you print to an 80-column monitor or a printer, there will be five per line.

Since this machine language program is one continuous block (that is, the program counter was assigned only once—in line 20), you can use the S command to save it to disk. Be sure there's a disk in the drive, then type S and the filename within quotes (for example, S "MESSAGE"), followed by a return. The S command saves in memory from the first program counter assignment (\$0B00 here) to the last program counter value at the end of assembly (\$0B1E for the original message).

SYNTAX OF SAR SOURCE CODE

The following terms are used in this text to refer to specific registers:

PC is short for program counter. In a machine language program, this is the address where the computer is working. In SAR, it's the address where the next assembled byte will be placed in memory.

A refers to the accumulator in the 6502 microprocessor. SAR uses the implied addressing mode for the accumulator. This is the same as in the built-in monitor, but different from some assemblers. Although SAR will let A be used as a label, it's bad practice to do

so! It could lead to strange errors when the accumulator addressing mode is used. If A has not been defined as a label, an instruction such as ROR A will generate an error.

.X refers to the X register within the 6502.

.Y refers to the Y register within the 6502.

SAR uses a flexible "floating format" for source code entry. Four fields are allowed, but not all are required. They are:

[LABEL] OPCODE ADR MODE [:REMARK]

or

[LABEL] PSEUDO-OP [DATA] [:REMARK]

First field labels are shown in lines 40, 50, 100, 110 and 120 of the sample program above. Any line can have a label *after* the PC is assigned. Only equates (as in line 30) are allowed before the PC is assigned, since SAR wouldn't know what location you wanted to use (there's *no* default). Labels should be followed by one space. SAR can handle more spaces, but they'll just make your source code bigger.

Standard opcodes are allowed, but A should *not* be used for the accumulator in the accumulator addressing mode. For example, if you want to shift the accumulator left, use ASL rather than ASL A.

Equates, which are used just like Basic variables, give an address or value to a label. If the address or value will be changed later, only the equate will have to be changed, then all label references will change automatically. The syntax for an equate is as follows:

LABEL=12345

or

LABEL=LABEL2

LABEL=LABEL2+\$3B

Addition, subtraction, multiplication and division to calculate an address are allowed to the right of the equal sign.

You don't need a space before or after the equal sign, but the space will make the L command list the equates in columns. Try several equates without spaces (for example, CB=\$0B00) to see the difference.

ADDRESS MODES

While the C-128's built-in monitor defaults to hex (base 16), most

other assemblers default to decimal (base 10). I prefer a default to hex, but I followed "the rest of the world" and used decimal.

SAR does *not* use the plus sign (+) to identify decimal numbers. This is different from the built-in monitor, but like most other assemblers. To use a decimal value in SAR, just write it like you usually would. For example, to load .A with a carriage return, or CHR\$(13), use LDA #13 (decimal) or LDA #\$D (hex). The plus sign is used by SAR for addition only. For example, STA LOC + 5 would add 5 to the value of the label LOC and store the contents of .A there.

Labels can have values from 0 to \$FFFF. During calculation of addresses, values up to \$FFFFFF (six digits) are valid, but the final result must be in the 0-\$FFFF range. Values higher or lower will result in an Illegal Address error.

Calculated addresses can include addition (+), subtraction (-), multiplication (*) and division (/). The * can also be used for the PC value at that line, if its position is correct. All calculations are handled from left to right. Any calculation involving the PC will "fill" to that address, if it's part of a relative PC assignment.

Be very careful with calculations involving labels defined later in the source. On pass 1, the PC value is used for such undefined labels, and one or two bytes are reserved according to the result of the calculation. If the labels Z1 and Z2 are defined after the calculation, a line such as LDA Z1-Z2 will assemble as LDA 0 on the first pass. It might produce a two-byte address after the labels are defined to their actual values. A Byte Count error would then result on pass 2. You can avoid this by using the force-absolute character (←). The same line written as LDA ←Z1-Z2 will assemble an absolute address on both passes. Define your labels before the calculation, if possible, by putting data tables before the calculation.

SAR uses standard syntax for all addressing modes except the accumulator, as explained above. SAR will use zero page addresses where possible. For this reason, define all zero page labels before assembly actually begins. Otherwise, two bytes will be reserved for the address on pass 1 but only one byte will be used on pass 2. This will result in a Byte Count error at the next column-one label (not an equate) or at the end of the second pass. The length of an address has no effect on how the address is evaluated. For example, if a

dummy address is used in assembly, use a dummy value of 256 or more. Here's an example:

```
90 ...  
100 STA TARGET+1  
110 STY TARGET+2  
120 TARGET LDA $0000,X ;DUMMY ADDRESS  
130 RTS  
140 ...
```

This will assemble line 120 as LDA \$00, and line 110 will overwrite the RTS value (line 130)! SAR will *not* detect this "error"! The correct way to write this line with SAR is:

```
120 TARGET LDA $1000 ;DUMMY 2 BYTE ADR
```

Since this value is greater than 255, two bytes are required for the address and the code will work as it was intended to. Since SAR ignores leading zeros, it's best to write your source without leading zeros, too—or at least write it as a "one-byte" address, if that's what you want. Be careful and be sure you understand this point, or you'll get a Byte Count error when you assemble your code!

You can force SAR to use an absolute (two-byte) address with the left-arrow (←) character before the address. This makes SAR use absolute mode no matter how small the address is. An example would be LDA ←\$15, which would make SAR produce code with an absolute address of \$0015 (two bytes). The left-arrow will work only with opcode addresses; it *won't* work with .BYTE.

The immediate address mode (#) allows an ASCII character or a high or low byte. ASCII characters must be enclosed in quotes. For example, LDA #"A" would load the accumulator with a value of \$41 (hex) or 65 (decimal).

To load a high or low byte, use the "greater than" (>) or "less than" (<) character, respectively. Here's an example:

```
520 ...  
530 LDX #<MSG1 ;GET LO BYTE  
540 STX TARGET+1 ;STORE IN LO ADR  
550 LDX #>MSG1 ;GET HI BYTE  
560 STX TARGET+2  
570 ...  
1000 TARGET LDA $FFFF,Y ;DUMMY ADR
```

A program such as this might print one of many messages in a

subroutine at line 1000. Lines 530 to 560 would change an address in the subroutine before it was called. The high and low bytes are often used to get bytes of an address to store in a vector, too. As always, the entire address is calculated before the high or low byte is found. If `MSG=$4FE` and the instruction `LDA#>MSG+15` are assembled, the resulting code will be `LDA #5` ($\$4FE + 15 = \$50D$). Note that without the dollar sign, the 15 is interpreted as decimal.

The PC, which keeps track of where the next byte assembled will be stored, can be changed in several ways. In SAR, it's represented by the `*` character. Before any assembly can occur, you *must* set the PC with an equate line, such as this:

```
20 *=$0B00 ;STORE IN 128 CASS. BUF
```

This line sets the value of the PC to `$0B00` (or 2816 decimal).

Usually the PC is assigned only once in this manner. A more common use for this command is to reserve space for a data block. That form uses a relative offset of the PC. Assume, for example, that you want to input a string and save it for later use. The following line will assign the value of the PC to the label `TEXT`, "assemble" five zero bytes starting at the current PC location, and add 5 to the value of the PC:

```
1500 TEXT *=*+5 ;RESERVE 5 BYTES
```

This same principle would work with a branch address, but you should avoid it. Here's a short example:

```
770 BMI *+4 ;COUNT BRANCH, OFFSET AND ALL BYTES TO  
    SKIP  
780 LDA #10  
790 JSR DO'IT ;BMI TARGET IS JSR
```

If any lines were added between 770 and 790, the offset (`*+4`) would have to be changed by hand. This defeats the purpose of an assembler! It's much better to put a label at 790 and use that label as the target for the branch in 770. If lines are added, the assembler will take care of the changes. (I use a label such as '790 in these cases.)

The PC can also be set to a label (if the label is assigned a value first) such as:

```
1250 *=LOC'2X
```

This type of assignment or a relative branch backwards (for example, `* = * - 10`) can't be used if a file is open for assembly to disk.

REMARKS

Using remarks in source code is very important. Like most other assemblers, SAR treats anything after a semicolon as a remark. Make remarks clear enough so you'll know what they mean next month or next year! The only time SAR won't ignore what follows a semicolon is when it's in text within quotes. Blank lines can be included for print formatting by putting just a semicolon on a line.

Since the source code is written as a "Basic" program with the C-128's screen editor, you can't use a question mark unless it's within quotes. If you do, it will be listed as PRINT. This is, I think, a small price to pay for the freedom allowed by the C-128's screen editor.

PSEUDO-OPS

Pseudo-ops are codes that tell the assembler to do something other than assemble normal opcodes. SAR uses only seven pseudo-ops, but can still do what most other assemblers can do.

SAR's pseudo-ops, which begin with a period (.) and are four letters long, are .BYTE, .WORD, .BANK, .CODE, .DISK, .NEXT and .LOOP.

.BYTE is followed by data—a table of values, messages, or just about anything else. You can also use it to store just the high or low byte of a label or number. Here's an example that shows how to use .BYTE:

```
750 DATA1 .BYTE 0,1,2,"A","B","C",>DATA2,>DATA3,<DATA2,<DATA3
```

This line stores the values 0, 1 and 2, the letters A, B and C (as CBM ASCII), the high bytes of labels DATA2 and DATA3, and, finally, the low bytes of DATA2 and DATA3. It can be shortened to:

```
750 DATA1 .BYTE 0,$102,"ABC",>DATA2,>DATA3,<DATA2,<DATA3
```

You can combine single-byte values (such as 1 and 2) into a two-byte hex value. This can save a comma and, if the data was in hex form, an extra \$ character. It's probably better to show each value as the first example does, and compress your data only if you have a very large program. The same method applies to text, but the memory savings are greater since two quotes and a comma can be saved. If it makes your code unclear, *don't* compress your data.

The reason why the 0 and 1 weren't combined into a two-byte

value is that SAR always chooses the shortest form of an address, and \$001 would be stored as just a single byte. The zero byte would *not* be stored.

.WORD To the 6502, a “word” is a two-byte address stored in low/high format. This is the standard way addresses are stored for opcodes with two-byte addresses. For example, JSR \$1234 is stored in memory as \$20 \$34 \$12, where the first byte is the value for a JSR instruction and the address follows in low/high order. Here are the two differences between **.WORD** and **.BYTE**:

1. The **.WORD** pseudo-op *always* stores two bytes of data for any number, while **.BYTE** uses a single byte if possible (that is, for values under 256).

2. The **.WORD** pseudo-op *always* stores the data in low/high order, while **.BYTE** stores the data as it was entered on the line.

Here’s an example that shows the **.WORD** pseudo-op:

```
100 LOC1=$1234
```

```
110 LOC2=$2001
```

```
120 LOC3=256
```

```
...
```

```
910 TABLE1 .WORD LOC1-1,LOC2-1,LOC3-1 ;STORE RTS'S IN  
TABLE
```

Line 910 will store the following bytes in memory in this order: \$33 \$12 \$00 \$20 \$FF \$00. Notice that all addresses are stored in low/high order. In this example, 1 is subtracted from the values before **.WORD** stores them. This is a common practice that pushes the address onto the stack later and “returns” a “fake” RTS to the address stored plus 1. This is, in effect, a programmable jump address (common in ROM applications).

Since **.WORD** is used specifically where a two-byte address is used, the high or low byte alone (> or <) or any form of string data will cause an error on a line with **.WORD**. In these cases, use the **.BYTE** pseudo-op instead.

.BANK SAR uses RAM bank 0 as the place to assemble code, unless directed to use something else. The other options are **.BANK 1** and **.DISK** (below). The **.BANK** pseudo-op requires a space followed by either a 0 or 1. Any other number or a missing space will produce an error.

If you must store your assembled code in the areas used by SAR (see SAR Memory Usage, below), you must use **.BANK 1** to store the code in memory. You probably *won't* be able to test your code

in bank 1 because of the bank switching involved on the C-128.

If you use the .BANK pseudo-op, the S command will produce a Syntax error, because SAR won't know where the code is stored and S was designed only for a continuous block in bank 0.

.CODE is used mainly for debugging or listing your final program. If the two-pass assembly is OK, a third pass is made to output your actual object code and source code together. The output can be sent to a printer or disk file by opening the file and using a CMD before doing the assembly. Oddly, lines containing the .CODE pseudo-op are not output, but are added at the beginning of the program.

Each time .CODE is found, object code output is toggled on or off. (At the start of each pass, this output is always off.) If you want to see only a section of your program, put .CODE at the beginning and end of the section you want to output and assemble as usual. If you're using linked source files, be sure to save the file before assembly!

Source lines that include a .BYTE or .WORD pseudo-op are listed followed by the actual bytes assembled. If a relative PC assignment is used to reserve memory (for example, `*=*+$15`), the code column will contain `[-FILL-]` to show that zeros are assembled in these locations. The pass for .CODE will be done before a .DISK pass if both are used in a source file.

.DISK saves code to disk as it's assembled. To assemble to a disk file (instead of to RAM) use the following syntax, where filename represents the name of your file:

```
790 .DISK "filename" ;OPEN FILE
```

To close the file (and assemble in RAM again), use .DISK without a filename. This is required before another file can be opened.

You can have as many disk files as you need, as long as each has a different name and one is closed before the next is opened. This is used mainly where several blocks are assembled at different locations in memory. Be sure to change the PC while the files are closed or a fatal error will result. Here's the correct way to assemble several blocks to disk:

```
10 ;ASSEMBLE MULTI-SEGMENT PROGRAM
20 *=$1B00 ;ASSEMBLE LO SECTION
30 LSTART .DISK "LO MEM FILE" ;OPEN FOR ASSEMBLY
40 ;... CODE GOES HERE
50 ;... ALL DONE
60 LO'END .DISK ;LABEL OPTIONAL/CLOSE FILE
```

```

70 *=$C000 ;SET PC FOR HI BLOCK
80 HSTART .DISK "HI MEM FILE" ;OPEN 2ND FILE
90 ;... CODE GOES HERE
100 ;...TILL DONE
110 ZZEND .DISK ;END LABEL/FINAL .DISK OPTIONAL

```

Of course, there's no code shown in this example, but it does show how to close the file, reset the PC and open the second file. The final .DISK (line 110) of any program is optional, and the last file will always be closed properly. The .DISK pseudo-op always assumes disk drive 8 and uses a file number of 8.

There are several problems that will prevent disk assembly:

1. Any error in assembly.
2. Any disk error.
3. Assigning the PC to a label with a file open (other than a forward relative assignment).

Note that any section assembled with .DISK *won't* be assembled to RAM on any pass. If the two-pass assembly is OK, the assembler will make an extra pass to do the actual disk assembly. This will prevent junk files on the disk; only disk drive errors can occur after the second pass.

.NEXT and .LOOP assemble source files that are too large to fit into memory. You can safely work with files up to about 175 blocks without these pseudo-ops.

If your file is larger than 175 blocks or you prefer to work with smaller source files, SAR will accept a chain of files. Each file should end with a .NEXT "filename" as the last line, and the filename given would be the name of the next file in the chain. The final file must use .LOOP "firstfile" to tell SAR which file to load before beginning another pass.

If a printer file is open (OPEN4,4:CMD4:A) during assembly, the load messages will be sent to the screen, but all other messages will be sent to the printer. The printer channel remains active after assembly is complete to let all the error messages be sent to the printer.

Be sure that any edited files are saved to disk *before* trying to assemble again! Any changes will be lost if they weren't saved.

EDITING SAR PROGRAMS

SAR'S L command will list the source in easy-to-read columns. However, editing these lines might take up more memory, because

the C-128 editor will ignore any leading spaces but accept spaces between fields. For this reason, it's best to list (with Basic) the line and then edit it. SAR can handle the extra spaces, but they'll use extra memory, extra disk space and take slightly longer to assemble.

LABELS

1. Labels can contain only digits (0-9), letters (A-Z) and the apostrophe.

2. Labels must *not* start with a number. A leading digit is assumed to be a decimal value.

3. Labels can contain a maximum of six characters. Longer labels will cause an error.

4. Labels can't contain a space.

5. A label can't be a valid opcode or pseudo-op. (Using the L command will show any opcode used as a label in the opcode column.)

6. A first-column label starting with a period is assumed to be a pseudo-op and will cause a fatal (stop-assembly) error. A valid pseudo-op without a period in the first column will cause a Pseudo-Op? error. If you want to use NEXT as a label, use NEXT1 or 'NEXT instead.

SAR MEMORY USAGE

SAR uses the following sections of bank 0 memory:

\$0200-\$029F Used as a buffer for the line being assembled.

\$10A0-\$10FB Used for bank switching and the like.

\$D000-\$E8A7 SAR's main program (machine language).

\$E8A8-\$FEFF Stores the label table during assembly.

There are also several addresses in zero page used by SAR, but these are used by Basic, too. Addresses \$03-\$08, \$24/25, \$3D/3E, \$60-\$68, \$7A and \$D0/D1 are used by SAR.

DIRECT MODE COMMANDS

SAR provides four direct-mode commands, wedged into the Basic Syntax Error routine. Each is the first letter of the command's function:

A assembles the source program currently in memory. The assembly can be stopped by pressing any key during a .DISK pass. If a fatal error occurs, assembly will stop with the error displayed. Fatal errors are those, such as No Origin, Bank and Disk, that could cause a system crash.

The source code for SAR is about 44K long, resulting in about 5.3K of assembled object code, which takes about 20 seconds to assemble on the first pass and another 13 seconds on the second pass. SAR always assembles in fast mode, so, if you're using 40-column mode, the screen will blank during assembly. However, you'll still hear the bell sound as each pass is finished. When assembly is complete (or a fatal error occurs), the original speed is restored and the 40-column screen turned on again.

If there are more errors than will fit on the screen, you have two choices: fix whatever errors you can see and assemble again or send all error messages to the printer. To do that, type the following line:

```
OPEN4,4:CMD4:A:PRINT#4:CLOSE4
```

If a fatal error occurs, the printer channel will be left open. Just type

```
PRINT#4:CLOSE4
```

and press return. Note that any SAR command can be mixed with Basic commands. See the Error Message section, below, for more information about specific causes of errors.

T lists all the labels (symbols) alphabetically in a table. You can press any key to stop a table listing.

L works with exactly the same syntax as Basic's List command. However, *L* prints a cleaner listing of the SAR source program.

S saves your assembled bank 0 object code to disk. You must give a filename in quotes after the *S* command, but the drive is always assumed to be 8. Any disk error will be reported, but a Drive Not Ready error might require you to turn the disk drive off and back on.

Any error in assembly will block the *S* command and result in a Syntax error if the *S* command is used. Using the *.DISK* or *.BANK* command will also block the *S* command and cause a Syntax error.

ERROR MESSAGES

SAR provides the following error messages:

Assembled Okay Congratulations! Your program assembled and is ready for testing.

Syntax Error A general error message. Examine the entire line for possible errors, such as missing spaces.

Label Not Found A pass-2 error caused by a label in an address field that was never defined. It could be a misspelled label.

No Origin A fatal pass-1 error. Tell SAR where to assemble the code and try again. The origin line might have another error that resulted in the PC not being set. This should be very easy to find.

Duplicate Label You've defined a single label more than once. The easiest way to find this is with the AFCO utility's Find command. Be sure that *all* the labels are corrected after this error!

Invalid Label Contains an invalid character or is too long. You might want to look for this label with AFCO to find and correct all instances of it.

Invalid Address Probably caused by a bad character in an address or a missing \$ character for a hex number. Remember that addresses can't exceed \$FFFF.

Table Overflow You should never see this! The table can contain over 700 labels, and SAR's source uses only about 465 of these.

Branch Range Error You tried to branch too far. You'll have to use a "relay" or a JMP instruction.

Opcode Error You have a bad opcode, probably a spelling error. All standard opcodes are supported.

Invalid Address Mode Examine your address mode for spelling errors, a colon starting a remark and other syntax errors in the line. If all looks in order, see if the address mode you're using is allowed with the opcode!

Disk File Error An error occurred while using a .DISK pseudo-op. If a filename is used, it may be too long. You can't open a second file if one is open.

Invalid Bank Error There's no space after .BANK, or a digit other than 0 or 1 is used for the bank number.

Loop Expected Error The last file in a chain of linked files didn't contain a .LOOP pseudo-op.

Pseudo-Op? A label in the first column is a valid pseudo-op but doesn't have a period in front of it. You probably used NEXT or LOOP as a label. Use the AFCO utility to change these to valid labels.

Byte Count Error There was an address defined or calculated as a single byte on pass 2 after two bytes were reserved for the address on pass 1. This would result in invalid label addresses. Always define zero page addresses at the beginning of your source code. This is a fatal error and is found at the time a first-column label doesn't match the address that was assigned to it on pass 1. The error occurred between the line given and the previous first-column label.

Break You hit a key during assembly. Any key but no-scroll will cause a break.

THE AFCO UTILITY

The AFCO utility was written to support SAR. It's a separate program so that if it's not needed, it won't use up low RAM memory. AFCO's code is stored at \$1504 to \$1AA3 in RAM bank 0. To use AFCO, just boot it or BLoad it into bank 0 and enter SYS 5380 to activate its wedge.

This program can Add (append) a program to the one in memory, Find or Change text in your source program, and Old (unNew) a program. These commands are explained below. AFCO can be used with or without SAR. AFCO commands must start in the first column and can't be mixed with Basic commands as SAR's commands can.

ADD will add a Basic program (from drive 8) to the program in memory and adjust the pointers as required. This facilitates subroutine libraries for easy addition to your source files. (It works well with Basic programs, too—with some planning!) To add a program, use the following syntax:

ADD"filename"

Line numbers aren't important to SAR, but you should renumber after all programs have been added.

FIND Use this command to find all occurrences of a string. It requires a delimiter before and after the text you're looking for. Quotes, periods and colons work best for delimiters. Use quotes only if you're looking for text within quotes. Use periods or colons if you want to find labels. If the source was edited on a C-64, you might have to use each (one at a time) to find all occurrences. (LOOP would not be tokenized on a C-64, but would be tokenized if edited on a C-128.)

CHANGE The syntax to change text is the same as for Find, but a second string follows the "find" string and there's a third delimiter. If the quote is used as a delimiter, a colon can be used as the middle delimiter to prevent the text from being tokenized. The syntax for Change is as follows:

CHANGE .PRINT.PRINT#4,.

or

CHANGE "TON:TIN"

The second form won't let the strings be tokenized, but the middle colon is accepted as a special delimiter. In any case, you'll be prompted Y/N/E for yes/no (change) or end for each string match found. If you want all strings changed (are you really sure!), just hold down the Y key. Use spaces in the text to avoid substrings. If you want to find IS, use " IS" to avoid finding THIS. Use common sense and caution with Change!

OLD If you accidentally "new" a program, the Old command will restore it for you. This must be done *before* any lines are added to the program (after the New command).

RUN it right: **C-64**

The 80-Column C-64

By Jay Taplin

I WOULDN'T TRADE MY C-64 for the world, but that loyalty doesn't keep me from wanting certain features available only on the "bigger computers."

A few months ago, I found myself wishing my 64 had an 80-column screen, so I wrote a program that would create it. While 80 Columns won't modify professional word processors to print 80-column characters, you can incorporate it easily into your own programs.

THE PROGRAM

The 80-column feature is achieved by using the high-resolution graphics screen, starting at memory location \$2000 (8192 decimal). The characters are created by taking every other bit value of the character and placing it in the high nibbles of memory locations \$2A7-\$2AD (679-685). Each location in that eight-byte section stands for one line of the character (0-7). Because the character values are accessed on request, there's no need for 2K of RAM to store the 80-column-character data, leaving more room for word processors and databases to store information.

I wrote 80 Columns in relocatable machine language and made

it accessible from either Basic with SYS commands or machine language with JSR\$ commands. The variables SE, CO, CL and PR stand for screen SEtup, COlor screen, screen CLear and PRint to screen, respectively; their values will change if you relocate the program by changing the value of BA in line 10 of the listing.

Screen Setup puts the computer in hi-res graphics mode. Just type SYS SE to access this function.

Color Screen sets the background to white and the foreground (text) to black. I found these the best colors to use on my monitor, as the smaller text is sometimes difficult to read. If you want to change the colors, choose a foreground color, multiply its value by 16, add the background color value, then store the sum at CO+3 (POKE CO+3, C).

Screen Clear—SYS CL—erases memory locations \$2000–\$3FFF (8192–16383) by erasing the graphics screen.

If you're a machine language programmer, note that Screen Setup starts on line 15, Screen Color on line 20 and Screen Clear on line 25. The main program starts on line 35.

The character printed is determined by the screen display codes (Appendix D in the *C-64 User's Guide* and Appendix B in the *Programmer's Reference Guide*). Poke (store) the value in \$FB (251). Also, to position the character on the screen, poke the value of X (0–79) in \$FC (252) and the value of Y (0–24) in \$FD (253). Once you've set all the values, enter SYS PR or JSR\$ PR, and the character should appear in the specified location.

THE DEMOS

To use the program, select 80 COLUMN from Menu 64. Once it's activated, run Menu 64 again and select DEMO1-80 to view the 80-column screen, containing 2000 characters. To view the regular 40-column screen, containing 1000 characters, enter RUN 40.

DEMO2-80 is a simple word processor that lets you type on the 80-column screen using the character, delete, return and cursor-right keys. F1 displays a menu with four options: Clear, Print, Load and Save. You can fill just one screen, because there's no scrolling. To quit the word processor, press run-stop/restore.

HINTS

1. Don't use 80 Columns as a replacement for the regular 40-column screen. I tried this, and the text gets very difficult to read after a while.

2. Test various colors with Color Setup. The colors I use may not be the best on your monitor. Also, changing the border color may help in viewing the characters.

3. Create interrupts to make some sections of the screen 80-column and others 40-column.

RUN it right: C-128 (in 80-Column mode)

Random Sound Experimenter

By Daniel Marcek

THE BASIC 7.0 SOUND COMMAND provides programmers with a rapid way to create sounds for games and other applications. However, getting the sound just right can be frustrating because of the number of variables involved. Random Sound Experimenter makes designing sounds easier, and it also provides a way to learn the workings and capabilities of the C-128's powerful Sound Interface Device (SID).

Refer to your *C-128 System Guide*, pages 129–137, for Sound command details.

To use Random Sound Experimenter, load and run Menu 128 in 80-column mode and select random sounds from the menu. Begin by selecting a sound, randomly generated by the program, that closely resembles the sound you want. Then fine-tune it by adjusting its parameters. You can also combine two or three sounds developed this way for an almost unlimited number of multiple-voice combinations.

The program is menu driven and runs in 80-Column mode. It begins by displaying a menu and initial voice parameters, then sounding voice 1. Parameters of this “active” voice are highlighted in yellow.

When playing voices simultaneously, an optional delay is available for postponing the start of any voice after the start of the previous

one. However, you must take care when sequencing sounds with such delays, because the total time (delay plus actual sounding of voices) is critical, and slight timing errors introduced by the program running in real time can throw it off. Fix any errors by fine-tuning the Duration and Delay parameters.

You can adjust all seven Sound command parameters and the voice delay (X) within their specified limits. Note that the voice delay is not included in the Sound command syntax and applies only within this program.

PROGRAM CONTROL

Following are the keys to press to activate various program functions:

Press 1, 2 or 3 to select an "active" voice. The chosen number will be highlighted on the screen, and the sound of that voice will be played.

Press 4 to toggle between playing a sequence of voices (see 5 below) just once or continuously. You can stop continuous play by pressing any key. The entire sequence of sounds and delays must finish before the stop will take effect.

Press 5 to play two or three voices in any order. Enter zero for voices not to be sounded.

Press 6 to replay voices selected with 5, above.

Press 7 to temporarily save the parameters of the "active" voice as a "prior" sound before pressing 8 (below) to generate new random sound. After pressing 8, you can quickly compare the two sounds by toggling between them within one voice. When you generate a random sound by pressing 8, all "active" parameters will be lost unless saved with under option 7.

Press 8 to generate a random sound. When you press 8, all previous parameters will be lost unless you save them by pressing 7 first (see above). The new sound will play for two seconds.

Press 9 to print out all "active" and "prior" sound parameters as a reference for future applications.

Press the escape key to leave Random Sound Experimenter and return to Basic.

Laser Math

By Tony Brantner

JUST WHEN YOU THOUGHT the computer skies were safe, another band of battle-crazed aliens is trying to conquer the earth. This new threat comes from a distant planet called Equato. The Equations, known throughout the universe for their savage brutality and poor table manners, must be stopped! To thwart them, you need more than luck and quick reflexes—you have to use your *brain*.

Laser Math combines arithmetic drill with the classic arcade theme of aliens invading the earth. Before the game begins, you're asked to select the math operation (addition, subtraction, multiplication or division) you'd like to practice, a skill level (from one, the easiest, to three) and a starting speed (from one, the slowest, to nine). The skill level you select determines the complexity of the problems randomly generated by the program. Load and run Menu 64, then select Laser Math to begin.

When the game screen appears, one to four spaceships carrying math problems advance toward your base. To shoot down a ship, type the answer to the problem it poses, then press the return key. If your answer is correct, a laser beam will destroy the ship. If you make a mistake typing in your answer, you can erase it using the delete key.

Occasionally the same answer may be correct for more than one problem. In that case, the spaceship closest to the base will be destroyed.

Blast 20 ships from the sky to advance to the next speed. The game ends if one of the ships lands on your base. Then you can press any key to return to the opening menu.

As you play, the number of hits and misses you've made are constantly displayed. Your score—plus two points for each hit and minus one for each miss—appears at the end of the game. The higher your score, the better you're getting at Laser Math! ■

RE_≡RUN

EDITOR-IN-CHIEF

DENNIS BRISSON

TECHNICAL EDITOR

TIM WALSH

MANAGING EDITOR

VINOY LAUGHNER

COPY EDITOR

PEG LePAGE

PROOFREADER/NEW PRODUCTS EDITOR

JANICE GREAVES

DESIGN AND LAYOUT

ANN DILLON

TYPESETTING

DEBRA DAVIES

FULFILLMENT CONSULTANT

DEBBIE BOURGAULT

9 Programs Included on this Disk

From the January/February RUN:

- ▶ Buttons
- ▶ Envelope Addresser
- ▶ Battling Boas

Plus: Extra Bonus Programs!

- ▶ Questionnaire
- ▶ Vanquish!
- ▶ SAR 128
- ▶ 80-Column 64
- ▶ Random Sound 128
- ▶ Laser Math

If any manufacturing defect becomes apparent, the defective disk will be replaced free of charge if returned by prepaid mail within 30 days of purchase. Send it, with a letter specifying the defect, to:

ReRUN • 80 Elm Street • Peterborough, NH 03458

Replacements will not be made if the disk has been altered, repaired or misused through negligence, or if it shows signs of excessive wear or is damaged by equipment.

The programs in ReRUN are taken directly from listings prepared to accompany articles in *RUN* magazine. They will not run under all system configurations. Use the RUN it Right information included with each article as your guide.

The entire contents are copyrighted 1990 by IDG Communications/Peterborough. Unauthorized duplication is a violation of applicable laws.

©Copyright 1990 IDG Communications/Peterborough

